**Analisi del Malware – Relazione Homework 3**

Studente: Andrea Pepe Matricola: 0315903

# Obiettivo

Analizzare il programma eseguibile “hw3.exe” utilizzando principalmente *Ghidra* e *OllyDbg* per determinare il codice di sblocco che rende funzionale il programma.

# Analisi di base

Il file eseguibile oggetto dell’analisi è in formato Portable Executable (PE), dunque un eseguibile per sistemi operativi Windows. Come prima operazione di analisi statica di base, è stata effettuata una ricerca delle stringhe utilizzate dal programma, in particolare usando il comando *strings* su sistema operativo Linux. Si riporta di seguito il risultato ottenuto, eliminando le stringhe non significative:

!This program cannot be run in DOS mode.

.text

P`data

P`.data

.rdata

0@.bss

.idata

.CRT

.tls

.rsrc

D$rTimef

D$v bef

D$zore

D$~Wind

ows

shut

down

D$@days

D$Ehour

D$Kminu

D$Otes

D$SUnlo

D$Wck c

D$[ode:

Escape from Windows v5 - M. Cesati, 2010, 2021

WARNING: there will be no shutdown without the proper unlock code!

0 seconds

%2ld seconds

Stop

Sorry, try again!

Shutdown time has come!

However, the unlock code is wrong.

If you want the full version of this wonderful tool,

you can get the unlock code for just ten bucks!

Ask to the nearest teacher around you!

EDIT

CreateWindow failed

BUTTON

SeShutdownPrivilege

InternalError

%s=%lu/0x%lx

DEBUG %s:%d

InternalError

Unknown error

\_matherr(): %s in %s(%g, %g) (retval=%g)

Argument domain error (DOMAIN)

Argument singularity (SIGN)

Overflow range error (OVERFLOW)

The result is too small to be represented (UNDERFLOW)

Total loss of significance (TLOSS)

Partial loss of significance (PLOSS)

Mingw-w64 runtime failure:

Address %p has no image-section

VirtualQuery failed for %d bytes at address %p

VirtualProtect failed with code 0x%x

Unknown pseudo relocation protocol version %d.

Unknown pseudo relocation bit size %d.

GCC: (tdm64-1) 9.2.0

AdjustTokenPrivileges

LookupPrivilegeValueA

OpenProcessToken

TextOutA

CloseHandle

CreateFileA

CreateFileMappingA

DeleteCriticalSection

EnterCriticalSection

ExitProcess

FatalAppExitA

GetCurrentProcess

GetCurrentProcessId

GetCurrentThreadId

GetFileInformationByHandle

GetLastError

GetModuleFileNameA

GetProcAddress

GetStartupInfoA

GetSystemTimeAsFileTime

GetTickCount

InitializeCriticalSection

IsDebuggerPresent

LeaveCriticalSection

LoadLibraryA

MapViewOfFile

QueryPerformanceCounter

SetUnhandledExceptionFilter

Sleep

TerminateProcess

TlsGetValue

UnhandledExceptionFilter

VirtualProtect

VirtualQuery

\_\_getmainargs

\_\_initenv

\_\_lconv\_init

\_\_p\_\_acmdln

\_\_p\_\_fmode

\_\_set\_app\_type

\_\_setusermatherr

\_amsg\_exit

\_cexit

\_initterm

\_iob

\_onexit

\_vsnprintf

abort

calloc

exit

fprintf

free

fwrite

malloc

memcpy

signal

strlen

strncmp

vfprintf

BeginPaint

CreateWindowExA

DefWindowProcA

DispatchMessageA

DrawTextA

EndPaint

ExitWindowsEx

GetClientRect

GetDlgItemInt

GetDlgItemTextA

GetMessageA

GetWindowLongA

KillTimer

LoadCursorA

LoadIconA

MessageBoxA

MoveWindow

PostQuitMessage

RedrawWindow

RegisterClassExA

SendDlgItemMessageA

SetDlgItemInt

SetDlgItemTextA

SetTimer

SetWindowLongA

ShowWindow

TranslateMessage

ADVAPI32.dll

GDI32.dll

KERNEL32.dll

msvcrt.dll

USER32.dll

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">

<trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">

<security>

<requestedPrivileges>

<requestedExecutionLevel level="asInvoker"/>

</requestedPrivileges>

</security>

</trustInfo>

<compatibility xmlns="urn:schemas-microsoft-com:compatibility.v1">

<application>

<!--The ID below indicates application support for Windows Vista -->

<supportedOS Id="{e2011457-1546-43c5-a5fe-008deee3d3f0}"/>

<!--The ID below indicates application support for Windows 7 -->

<supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2440225f93a}"/>

<!--The ID below indicates application support for Windows 8 -->

<supportedOS Id="{4a2f28e3-53b9-4441-ba9c-d69d4a4a6e38}"/>

<!--The ID below indicates application support for Windows 8.1 -->

<supportedOS Id="{1f676c76-80e1-4239-95bb-83d0f6d0da78}"/>

<!--The ID below indicates application support for Windows 10 -->

<supportedOS Id="{8e0f7a12-bfb3-4fe8-b9a5-48fd50a15a9a}"/>

</application>

</compatibility>

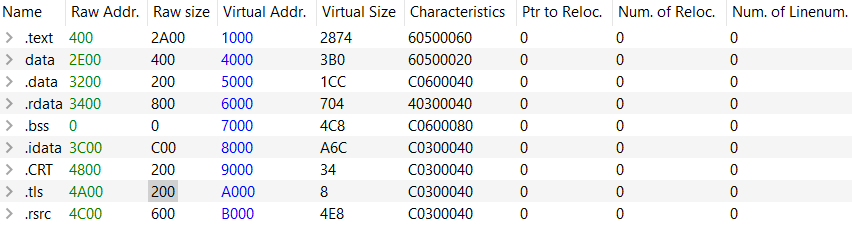
</assembly>

Dalla stringa “GDI32.dll” e dal nome di alcune API come, ad esempio, “GetMessageA” e “TranslateMessage” si desume che il programma abbia una GUI.

Inoltre, sono molto interessanti i nomi di alcune API che potrebbero essere state usate per ostacolare il reversing del codice del programma, adottando misure anti-debugging; in particolare: *GetTickCount, IsDebuggerPresent, LoadLibraryA, GetProcAddress, QueryPerformanceCounter, SetUnhandledExceptionFilter, UnhandledExceptionFilter.*

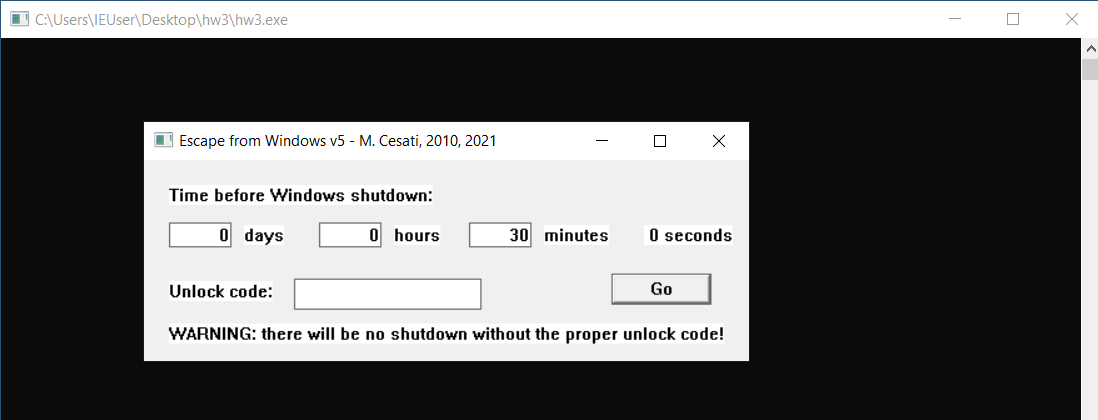
Infine, viene stampato tra le stringhe anche il contenuto di un file manifest in XML che descrive la compatibilità dell’eseguibile con le diverse versioni dei sistemi operativi Windows: sono supportati Windows Vista, 7, 8, 8.1 e 10. Infatti, usando il software *Resource Hacker*, tale file manifest risulta essere presente nella sezione dedicata alle risorse del file eseguibile.

Si è passati ad analizzare le testate del file eseguibile tramite l’utilizzo di *PE-Bear*: dalle informazioni riguardanti i SECTION\_HEADERS si è confrontata la raw size e la virtual size di ogni sezione. Sostanzialmente, i valori erano quasi sempre simili, fatta eccezione per la sezione .tls: infatti, come visibile dallo stesso *PE-Bear*, sono presenti nella TLS directory i riferimenti a 2 tls callbacks. Esse potrebbero essere usate per eseguire del codice prima che il debugger riesca a fermarsi sull’entry point del programma.



Pe-Bear: analisi dei Section Hdrs

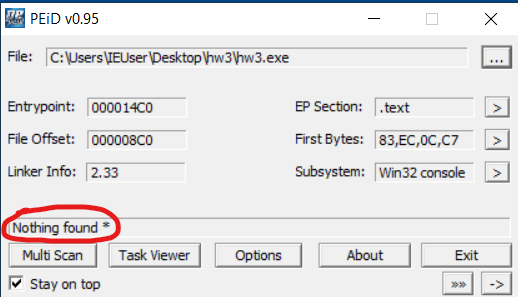
Si è proceduto con l’analisi dinamica di base e come prima cosa è stato lanciato l’eseguibile su una macchina virtuale con sistema operativo Windows 10 per osservarne il comportamento: viene aperta una finestra per programmare lo shutdown di Windows; non inserendo alcun codice di sblocco, allo scadere del countdown, appare una message box con un messaggio d’errore e il programma termina senza effettuare lo spegnimento del sistema.



Esecuzione del programma

Analizzando il processo in esecuzione tramite *ProcessExplorer* e *ProcessMonitor,* si nota che vengono effettuate diverse operazioni sul registro di sistema, ma non sembrano significative. Utilizzando *RegShot*, per fare un confronto tra lo status del registro di sistema prima di lanciare l’eseguibile e quello dopo averlo lanciato, se ne ha la conferma.

Un’analisi tramite *PEiD*, conferma che l’eseguibile non è stato compresso o offuscato tramite l’uso di un packer, così come era desumibile anche dalle informazioni ottenute usando *PE-Bear*:



Analisi con PEiD

Provando a lanciare il programma all’interno di *OllyDbg*, si nota che la finestra per programmare lo spegnimento di Windows non viene mostrata e, anziché rimanere running eseguendo il message loop, il programma termina. È evidente che sono presenti delle misure anti-debugging. Ci si pone quindi come obiettivo quello di individuare tali misure tramite l’uso dell’analisi statica avanzata con *Ghidra* combinata con *OllyDbg*, in maniera tale da inibirle e poter così effettuare l’analisi dinamica avanzata con il debugger.

# Analisi avanzata

Cercando nella sezione degli import l’API TranslateMessage() e guardandone i riferimenti, è stato individuato il message loop e quini la funzione *WinMain:*



Decompilato della funzione WinMain() in cui figurano già informazioni individuate in un secondo momento

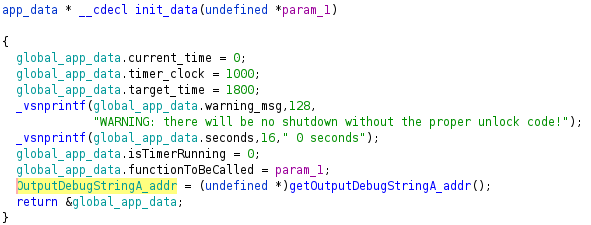
Ciò ha permesso di individuare facilmente la Window Procedure, ridenominata *WndProc.* Tuttavia, prima di inizializzare la struttura dati WNDCLASSEXA, viene invocata una funzione, la quale è stata ridenominata *getFileInfo*. In questa funzione, tramite l’API *GetModuleFilenameA,* si ottiene il path del file eseguibile del processo corrente. Se tale path esiste, si ottiene un handle aperto al file tramite l’invocazione a *CreateFileA,* che viene usato come parametro nella chiamata a *GetFileInformationByHandle*. In questo modo si ottengono in una struttura dati informazioni sul file e, in particolare, viene salvata in una variabile globale ridenominata *nSizeFileLow* la dimensione in bytes del file. A questo punto, viene invocata l’API *CreateFileMappingA* per creare ed ottenere un handle ad un file mapping object per il file eseguibile. Infine, tramite la *MapViewOfFile*, viene mappato la vista del file mapping object nell’address space del processo chiamante e il valore di ritorno, che corrisponde all’indirizzo iniziale della vista mappata, viene salvato in una variabile globale, ridenominata *startingAddress.*

È riportato di seguito il decompilato della funzione appena descritta:



Decompilato della funzione getFileInfo()

In seguito, nella *WinMain,* viene invocata una funzione (*init\_data*) che svolge il compito di inizializzare la struttura dati fondamentale contenente i dati dell’applicazione. Dunque, analizzandola, si individuano alcuni campi di tale struttura dati. Tuttavia, oltre a fare ciò, invoca un’altra funzione, la quale usa *LoadLibraryA* per caricare la dll kernel32.dll e *GetProcAddress* per ottenere l’indirizzo dell’API *OutputDebugStringA.* Tale indirizzo viene salvato in una variabile globale.



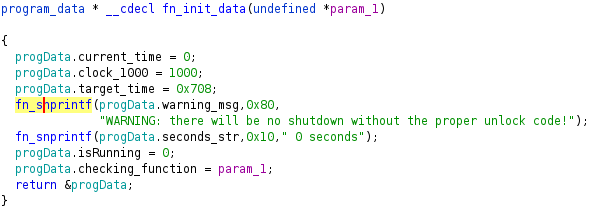
Decompilato della funzione init\_data()

L’analisi della funzione *getOutputDebugStringA\_addr()* è stata complicata dall’utilizzo di misure anti-disassembling; in particolare, è stata utilizzata la tecnica del disassemblaggio impossibile con il seguente schema:

**// TODO: schema bytes EB FF C0 48 (jump -1, inc eax, dec eax)**

In tale funzione, viene invocata la RegisterClassExA per registrare una classe di finestra: dunque, identificando la struttura di tipo WNDCLASSEX, il cui indirizzo viene passato come parametro alla API, è stato possibile individuare tra i campi, con l’ausilio del decompilatore, il class name della finestra, che corrisponde a “W04”, e l’indirizzo della WindowProc incaricata della gestione degli eventi inviati alla finestra, ridenominata *WndProc*

In seguito, viene invocata la CreateWindowExA per creare la finestra dell’applicazione ed ottenerne un handle. A tale API viene passato come ultimo parametro (lpParam) il valore di ritorno di una funzione chiamata appena prima e ridenominata *fn\_init\_data*, in quanto si presume che inizializzi una struttura contenente i dati dell’applicazione. Oltre a scrivere dei valori interi in una struct globale presente nella sezione dati, inserisce anche delle stringhe in alcuni buffer, usando una funzione che con buone probabilità corrisponde alla snprintf e che dunque viene ridenominata *fn\_snprintf*. Si riporta di seguito il decompilato della funzione *fn\_init\_data*, ma in cui figurano già dei riferimenti più chiari alla struttura dati in questione che sono stati definiti in un secondo momento.



In effetti, il parametro lpParam della CreateWindowExA rappresenta un puntatore ad un valore passato alla finestra tramite il campo lpCreateParams di una struttura di tipo CREATESTRUCT e puntato dal parametro lParam del messaggio WM\_CREATE, che la finestra gestisce al momento della sua creazione. Tale parametro contiene per l’appunto dati addizionali user-defined da associare alla finestra quando viene creata, con ogni probabilità raggruppati in una struct.

# Analisi della WndProc

Quindi, si è passati ad analizzare la *WndProc* per desumerne informazioni utili a ricostruire la struttura dati in questione, da integrare con le informazioni già ottenute in precedenza dalla *fn\_init\_data*. Innanzitutto, è stato utilizzato il tool function graph per suddividere la funzione in blocchi logici e raggrupparli in base alla gestione del tipo di messaggio ricevuto dalla finestra; la *WndProc* gestisce in maniera non di default i messaggi:

* WM\_SIZE;
* WM\_PAINT;
* WM\_COMMAND;
* WM\_CREATE;
* WM\_DESTROY.

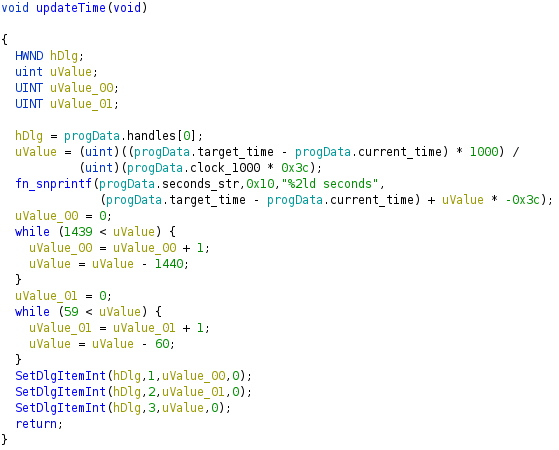
Come prima operazione, prima di discernere il tipo di messaggio ricevuto dalla finestra, viene invocata l’API GetWindowLongA passandole come secondo parametro la macro GWL\_USERDATA. In questo modo si ottiene un riferimento alla struttura dati passata alla finestra con la SetWindowLongA al momento della creazione della finestra, il quale viene salvato in una variabile locale.

Analizzando la parte della funzione che gestisce il messaggio WM\_CREATE, si identifica un array di 6 HANDLE inserito come campo della struttura dati che si sta cercando di ricostruire, ridenominata *program\_data.* Tale array, cui è stato dato il nome di *handles* è così composto:

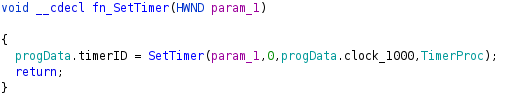
* handles[0] 🡪 hWnd, handle alla finestra principale del programma;
* handles[1], handles[2], handles[3] 🡪 Sono gli handles alle edit boxes corrispondenti rispettivamente ai campi giorni, ore e minuti per l’input dell’utente;
* handles[4] 🡪 handle alla finestra di tipo bottone per far partire/fermare il countdown;
* handles[5] 🡪 handle alla edit box in cui scrivere il codice di sblocco.



Come ultime operazioni nella gestione del messaggio WM\_CREATE, vengono invocate due funzioni: la prima, ridenominata *updateTime,* è utilizzata per cambiare i valori di giorni, ore, minuti e secondi rimanenti allo spegnimento. In questo contesto, essendo chiamata per la prima volta, setta tali valori a quelli impostati dalla *fn\_init\_data*, che corrispondono a 30 minuti (0x708 = 1800 secondi). Grazie all’analisi di questa funzione, si riesce a dare un significato ai valori dei campi della struct *program\_data* assegnati dalla funzione di inizializzazione; in particolare si riconosce un campo che conta il tempo trascorso (*current\_time*), uno che identifica il tempo target per lo spegnimento (*target\_time*) e uno che rappresenta la lunghezza del clock (*clock\_1000*), pari a 1000 (millisecondi, ovvero un secondo). Inoltre, viene identificato un campo che consiste in un buffer di 16 bytes (*seconds\_str*) contenente la stringa “%2ld seconds” che viene aggiornata periodicamente al passare di ogni secondo.



La seconda funzione è la *fn\_SetTimer* incontrata all’inizio dell’analisi, che non fa altro che invocare l’API SetTimer per creare un timer associato alla finestra del programma, specificando come valore del clock quello inserito nella struct *program\_data* e indicando la TIMERPROC da eseguire per processare messaggi di tipo WM\_TIMER. Il valore di ritorno della SetTimer viene assegnato ad un campo della struct globale individuata in precedenza; dunque, tale campo sarà il *timerID* .

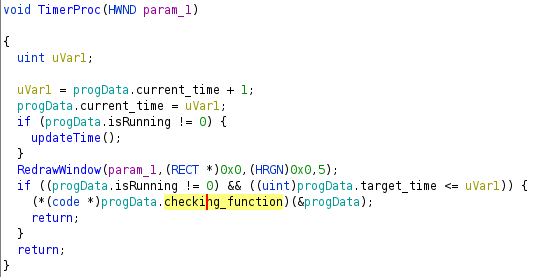


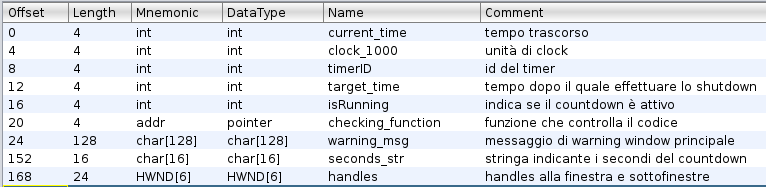
# Struttura Dati e Codice di Sblocco

A questo punto dell’analisi, si è tornati sulla funzione *TimerProc* e, con le informazioni ottenute sulla struttura dati usata dal programma, ci si rende conto che la CALL all’indirizzo nel segmento dati trovata inizialmente è una CALL ad un campo della struct. Dunque, tale campo contiene l’indirizzo di una funzione, alla quale viene passato come parametro un riferimento alla struttura dati globale. Per indagare su quale funzione fosse, si ritorna indietro alla *WinMain*, che passa come parametro alla *fn\_init\_data* una variabile globale. Tale variabile viene attribuita al campo della struct di cui si effettua la call nella *TimerProc*.

A questo punto, la struttura dati *program\_data* usata dall’applicazione è ricostruita. Con ogni probabilità, la funzione che viene invocata è quella che effettua il controllo sulla correttezza del codice di sblocco ed esegue lo shutdown di Windows.

Si riportano di seguito il decompilato della *TimerProc* e la struttura dati *program\_data* definita in *Ghidra,* dopo averne denominato il campo che punta alla funzione come “checking\_function”:

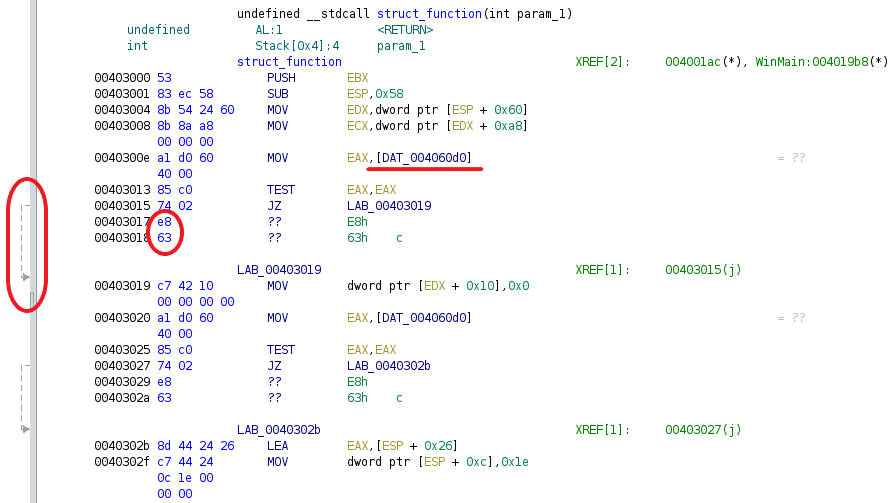




Ora, non resta altro che indagare sulla funzione che viene invocata dalla *TimerProc* e di cui la struct conserva l’indirizzo. Si nota che tale funzione è presente anch’essa nel segmento dati. *Ghidra* non è riuscito a riconoscerla come funzione e quindi i bytes delle istruzioni che la compongono non sono stati disassemblati. Si forza *Ghidra* a disassemblare e a riconoscere la funzione; il risultato ottenuto è il seguente (solo la parte iniziale della funzione è mostrata di seguito):



Come si può notare, vengono segnalate delle istruzioni CALL in rosso, ad indicare che gli indirizzi specificati non sono presenti all’interno del file eseguibile. Analizzando le informazioni fornite dal disassembler di Ghidra, si nota che prima di ognuna di queste CALL, c’è un’istruzione JZ ad una label al terzo byte dei 5 che compongono l’istruzione di CALL successiva (1 byte per l’opcode + 4 bytes per l’indirizzo a 32 bit a cui saltare). Si desume che, molto probabilmente, il disassemblatore vada in confusione e quindi si decide di disassemblare il codice poco alla volta, per blocchi, seguendo le indicazioni fornite dalle labels. Il risultato ottenuto è il seguente:



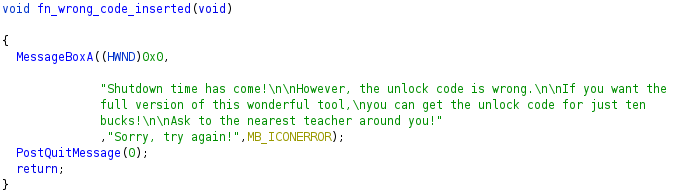
Nell’immagine, è evidenziata una coppia di bytes [E8, 63] che si ripetono di frequente nella funzione, ma che non rappresentano nessuna istruzione; bensì confondevano il disassemblatore di *Ghidra*, inducendolo ad interpretarli come facenti parte di una istruzione di CALL, in quanto l’opcode di tale istruzione è appunto il byte E8. Si nota inoltre che il flusso d’esecuzione giunge a tali bytes solo se il valore della variabile globale DAT\_004060d0 è diverso da zero. Tuttavia, tale variabile è situata nel segmento BSS, dunque è inizializzata a 0; inoltre, non viene mai riferita in scrittura e il suo valore non verrà mai modificato, risultando sempre pari a zero. In conclusione, il salto condizionale corrispondente all’istruzione JZ prima dei due bytes [E8, 63] verrà sempre preso e il flusso di esecuzione non si ritroverà mai ad eseguire tali bytes.

Si suppone che tale struttura sia stata ottenuta utilizzando un offuscatore di codice per ostacolare il reversing del programma.

Si decide di effettuare per ogni occorrenza della coppia di bytes [E8, 63] una Patch Instruction all’interno di *Ghidra*, inserendo una coppia di NOP [90, 90]. A questo punto, la funzione, ridenominata *struct\_method,* diventa molto più leggibile sfruttando il decompilatore. Nell’immagine che si riporta di seguito, figura già una chiamata alla API AdjustTokenPrivileges. In realtà, essa non era stata riconosciuta da *Ghidra* ed è stato necessario effettuare una ulteriore forzatura a disassemblare dopo averne pulito il codice erroneamente disassemblato in precedenza. A questo punto, ne è venuta fuori una semplice istruzione di jump alla API AdjustTokenPrivileges della DLL ADVAPI32 ed è quindi stata indicata come una thunk function. Allo stesso modo, è stato necessario disassemblare la funzione indicata come *fn\_wrong\_code\_inserted*, che fa apparire la finestra contenente il messaggio d’errore quando il codice di sblocco inserito dall’utente risulta essere sbagliato.



Dunque, la *struct\_method* chiama la GetDlgItemTextA per ottenere l’input inserito dall’utente nella edit box del codice di sblocco (5° HANDLE nell’array *handles* della struttura *program\_data*). Dopodiché, controlla che la lunghezza di tale input sia pari a 9 e, char per char, effettua il confronto con il codice di sblocco corretto, il quale risulta essere la stringa “3RnESt0!?”. Se il controllo non va a buon fine, viene invocata la *fn\_wrong\_code\_inserted* per mostrare il messaggio d’errore:



Invece, qualora il codice di sblocco inserito fosse corretto, la funzione si prepara ad effettuare lo shutdown del sistema nel seguente modo:

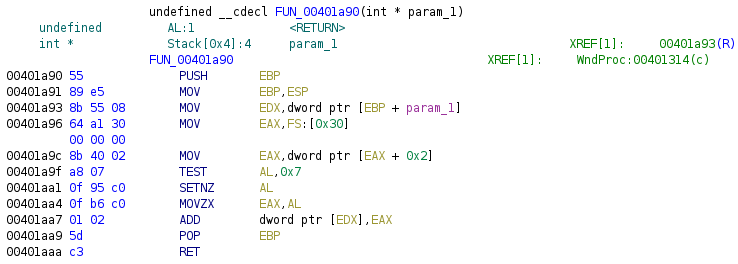
1. Chiama l’API GetCurrentProcess per ottenere uno pseudo handle per il processo corrente;
2. Invoca la OpenProcessToken per aprire un access token associato con il processo corrente, specificando una access mask pari a 0x28, ovvero l’or delle macro TOKEN\_ADJUST\_PRIVILEGES (0x20) e TOKEN\_QUERY (0x08). Non essendo presenti sulla documentazione Microsoft, tali valori sono stati reperiti al seguente link: <https://www.pinvoke.net/default.aspx/advapi32.openprocesstoken>. Tali macro indicano che è possibile effettuare delle query sull’access token e che se ne possono abilitare o disabilitare i privilegi;
3. Effettua una chiamata all’API LookupPrivilegeValueA con la quale recupera il Locally Unique Identifier (LUID) del privilegio SE\_SHUTDOWN\_NAME, identificato dalla stringa “SeShutdownPrivilege”, e lo salva nel campo Privileges di una struct locale di tipo TOKEN\_PRIVILEGES. In questo modo, ha ottenuto l’identificatore sulla macchina locale del privilegio per effettuare lo shutdown del sistema;
4. Il campo Privileges della struct TOKEN\_PRIVILEGES è a sua volta una struct, di tipo LUID\_AND\_ATTRIBUTES, che ha come primo campo il LUID del privilegio e come secondo campo un attributo, che viene settato a 2, ovvero a SE\_PRIVILEGE\_ENABLED;
5. L’indirizzo della struct TOKEN\_PRIVILEGES viene passato, insieme all’handle al token ottenuto con la OpenProcessToken, alla API AdjustTokenPrivileges per abilitare il privilegio necessario ad effettuare lo shutdown per l’access token appena aperto;
6. Se tutto è andato a buon fine, viene effettuato lo spegnimento del sistema invocando l’API ExitWindowsEx, passando come primo parametro 5 e come secondo parametro 0. Il primo parametro corrisponde all’or dei due flag EWX\_SHUTDOWN (0x01) ed EWX\_FORCE (0x04): ciò vuol dire che il sistema viene arrestato forzando tutte le applicazioni a terminare, comportando una potenziale perdita di dati da parte delle applicazioni. Il secondo parametro, pari a 0, indica che non viene indicato alcun motivo per l’inizio dello shutdown.

# Contromisure anti-debugging

Come ultima operazione nel processo di analisi e reversing dell’eseguibile, si ritorna su una funzione chiamata all’interno della *WinProc* inizialmente ignorata:



A questa funzione FUN\_00401a90, viene passato come parametro di ingresso l’indirizzo di uMsg, ovvero l’argomento della funzione *WinProc* che rappresenta il codice identificativo del messaggio ricevuto dalla finestra. Di seguito viene riportato il disassemblato della funzione:



In sistemi con architettura x86, il registro FS è usato per accedere tramite offset ad una struttura dati detta TIB (Thread Information Block), o anche TEB (Thread Envirnment Block), contenente informazioni sul thread attualmente in esecuzione. In particolare, con FS:[0x30], ovvero con offset 0x30 a partire da FS, si ottiene l’indirizzo di una ulteriore struttura dati detta PEB (Process Environment Block), come suggerito dalla pagina Wikipedia: <https://en.wikipedia.org/wiki/Win32_Thread_Information_Block>.

La struttura dati PEB contiene informazioni sul processo corrente, ma non è completamente documentata da Windows. Tuttavia, la funzione tenta di recuperare il valore ad offset 2 di tale struttura, che, come desunto dalle risorse web <https://www.aldeid.com/wiki/PEB-Process-Environment-Block/BeingDebugged> e <https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb>, corrisponde al campo BeingDebugged. Tale campo è un byte che è pari a 1 nel caso in cui il processo corrente è eseguito in modalità di debug, 0 altrimenti.

Dunque, ciò che la funzione chiamata dalla *WndProc* fa, è di consultare tale valore e, nel caso in cui l’applicazione sia stata lanciata con un debugger, modifica il valore di uMsg aumentandolo di 1 prima che ne venga fatto il confronto con le varie macro corrispondenti ai tipi di messaggi che la finestra deve gestire. Altrimenti, il valore di uMsg resta inalterato. In questo modo, si ostacola l’analisi dinamica avanzata del programma, facendogli assumere un comportamento completamente diverso quando eseguito tramite un debugger.

# Conclusione

In conclusione, il codice di sblocco del programma è “3RnESt0!?”. Esso è stato anche testato su macchina virtuale ed ha riportato il risultato atteso, ovvero lo spegnimento del sistema. Il sistema viene spento forzando tutte le applicazioni attive a terminare, causando una possibile perdita di dati delle applicazioni, e senza specificare la ragione dello shutdown.

L’applicazione usa una struttura dati un po’ “anomala”, in quanto è una struct che ha un campo contenente l’indirizzo di una funzione, facendo pensare ad una classe in un contesto Object Oriented. Inoltre, tale funzione è contenuta nella sezione dati dell’eseguibile e probabilmente è stata modificata utilizzando delle tecniche di offuscamento per complicarne il reversing.

Infine, si riporta che l’applicazione tenta di proteggersi da un’analisi effettuata con un debugger, rilevando tale situazione e mutando il suo comportamento a runtime, così da ingannare l’analista.